**Static Single Assignment Form**

**Last Time**
- Static single assignment (SSA) form

**Today**
- Applications of SSA

---

**Dead Code Elimination for SSA**

**Dead code elimination**

```plaintext
while 3 a variable v with no uses and whose def has no other side effects
   Delete the statement s that defines v
   for each of s’s uses w
      Delete the use from list of uses of variable w
```

If y becomes dead and there are no other uses of x, then the assignment to x becomes dead, too

- Contrast this approach with one that uses liveness analysis
  - This algorithm updates information incrementally
  - With liveness, we need to invoke liveness and dead code elimination iteratively until we reach a fixed point
Implementing Simple Constant Propagation

**Standard worklist algorithm**
- Identifies simple constants
- For each program point, maintains one constant value for each variable

**Problem**
- Inefficient, since constants may have to be propagated through irrelevant nodes

**Solution**
- Exploit a sparse dependence representation (e.g., SSA)

---

Sparse Simple Constant Propagation

**Reif and Lewis algorithm**
- Identifies simple constants
- Faster than Simple Constants algorithm

**SSA edges**
- Explicitly connect defs with uses
- How would you do this?

**Main Idea**
- Iterate over SSA edges instead of over all CFG edges
Sparse Simple Constants Algorithm (Ch. 19 in Appel)

worklist = all statements in SSA

while worklist ≠ ∅

  Remove some statement S from worklist
  if S is x = phi(c,c,...,c) for some constant c
     replace S with v = c
  if S is x=c for some constant c
     delete S from program
     for each statement T that uses x
        substitute c for x in T
     worklist = worklist union {T}

Copy Propagation

Algorithm
worklist = all statements in SSA
while worklist ≠ ∅
  Remove some statement S from worklist
  if S is x = phi(y) or x = y
     for each statement T that uses x
        replace all use of x with y
  worklist = worklist union {T}
  delete S
Reuse Optimization

Idea
– Eliminate redundant operations in the dynamic execution of instructions

How do redundancies arise?
– Loop invariant code (e.g., index calculation for arrays)
– Sequence of similar operations (e.g., method lookup)
– Same value be generated in multiple places in the code

Types of reuse optimization
– Value numbering
– Common subexpression elimination
– Partial redundancy elimination

Local Value Numbering

Idea
– Each variable, expression, and constant is assigned a unique number
– When we encounter a variable, expression or constant, see if it’s already been assigned a number
  – If so, use the value for that number
  – If not, assign a new number
– Same number ⇒ same value

Example
a := b + c
b := a
d := b

e := d + c

b → #1 #3
c → #2
b + c is #1 + #2 → #3
a → #3
d → #1
d + c is #1 + #2 → #3
e → #3
Local Value Numbering (cont)

Temporaries may be necessary

\[
\begin{align*}
    a & := b + c \\
    a & := b \\
    d & := a + c
\end{align*}
\]

\[
\begin{align*}
b & \rightarrow \#1 \\
c & \rightarrow \#2 \\
b + c & \text{ is } \#1 + \#2 \rightarrow \#3 \\
a & \rightarrow \#1 \\
a + c & \text{ is } \#1 + \#2 \rightarrow \#3 \\
d & \rightarrow \#3
\end{align*}
\]

\[
\begin{align*}
t & := b + c \\
a & := b \\
d & := b + c
\end{align*}
\]

\[
\begin{align*}
b & \rightarrow \#1 \\
c & \rightarrow \#2 \\
b + c & \text{ is } \#1 + \#2 \rightarrow \#3 \\
t & \rightarrow \#3 \\
a & \rightarrow \#1 \\
a + c & \text{ is } \#1 + \#2 \rightarrow \#3 \\
d & \rightarrow \#3
\end{align*}
\]

Global Value Numbering

How do we handle control flow?

\[
\begin{align*}
w & = 5 \\
x & = 5 \\
w & \rightarrow \#1 \\
x & \rightarrow \#1
\end{align*}
\]

\[
\begin{align*}
w & = 8 \\
x & = 8 \\
w & \rightarrow \#2 \\
x & \rightarrow \#2
\end{align*}
\]

\[
\begin{align*}
y & = w + 1 \\
z & = x + 1
\end{align*}
\]
Global Value Numbering (cont)

Idea [Alpern, Wegman, and Zadeck 1988]
- Partition program variables into congruence classes
- All variables in a particular congruence class have the same value
- SSA form is helpful

Approaches to computing congruence classes
- Pessimistic
  - Assume no variables are congruent (start with $n$ classes)
  - Iteratively coalesce classes that are determined to be congruent
- Optimistic
  - Assume all variables are congruent (start with one class)
  - Iteratively partition variables that contradict assumption
  - Slower but better results

Role of SSA Form

SSA form is helpful
- Allows us to avoid data-flow analysis
- Variables correspond to values

\[
\begin{align*}
\text{a} & \not\equiv \text{anything} \\
a = b & \\
... & \\
a = c & \\
... & \\
a = d & \\
\end{align*}
\]

\[
\begin{align*}
a_1 = b & \\
... & \\
a_2 = c & \\
... & \\
a_3 = d & \\
\end{align*}
\]

Congruence classes: \{a_1, b\}, \{a_2, c\}, \{a_3, d\}
Basis

Idea

– If \( x \) and \( y \) are congruent then \( f(x) \) and \( f(y) \) are congruent

\[
\begin{align*}
x \text{ and } y \text{ are congruent} \\
\begin{array}{l}
ta = a \\
tb = b \\
x = f(a, b) \\
y = f(ta, tb)
\end{array}
\end{align*}
\]

– Use this fact to combine (pessimistic) or split (optimistic) classes

Problem

– This is not true for \( \phi \)-functions

\[
\begin{align*}
a_1 &= x_1 \\
a_2 &= y_1 \\
b_1 &= x_1 \\
b_2 &= y_1 \\
a_3 &= \phi_n(a_1, a_2) \\
b_3 &= \phi_m(b_1, b_2)
\end{align*}
\]

Solution: Label \( \phi \)-functions with join point

Pessimistic Global Value Numbering

Idea

– Initially each variable is in its own congruence class
– Consider each assignment statement \( s \) (reverse postorder in CFG)
  – Update LHS value number with hash of RHS
  – Identical value number \( \Rightarrow \) congruence

Why reverse postorder?

– Ensures that when we consider an assignment statement, we have already considered definitions that reach the RHS operands

\[
\text{Postorder: } d, c, e, b, f, a
\]
Algorithm

for each assignment of the form: “x = f(a, b)”
ValNum[x] ← UniqueValue() // same for a and b

for each assignment of the form: “x = f(a, b)” (in reverse postorder)
ValNum[x] ← Hash(f ⊕ ValNum[a] ⊕ ValNum[b])

Snag!

Problem
– Our algorithm assumes that we consider operands before variables that depend upon it
– Can’t deal with code containing loops!

Solution
– Ignore back edges
– Make conservative (worst case) assumption for previously unseen variable (i.e., assume its in it’s own congruence class)
Optimistic Global Value Numbering

Idea
– Initially all variables in one congruence class
– Split congruence classes when evidence of non-congruence arises
  – Variables that are computed using different functions
  – Variables that are computed using functions with non-congruent operands

Splitting

Initially
– Variables computed using the same function are placed in the same class

\[
x_1 = f(a_1, b_1) \\
\ldots \\
y_1 = f(c_1, d_1) \\
\ldots \\
z_1 = f(e_1, f_1)
\]

Iteratively
– *Split* classes when corresponding operands are in different classes
– Example: assume \( a_1 \) and \( c_1 \) are congruent, but \( e_1 \) is congruent to neither
Splitting (cont)

Definitions
- Suppose P and Q are sets representing congruence classes
- Q splits P for each i into two sets
  - \( P \setminus Q \) contains variables in P whose \( i^{th} \) operand is in Q
  - \( P / Q \) contains variables in P whose \( i^{th} \) operand is not in Q
- Q properly splits P if neither resulting set is empty

\[
\begin{align*}
x_1 &= f(a_1, b_1) \\
& \quad \cdots \\
y_1 &= f(c_1, d_1) \\
& \quad \cdots \\
z_1 &= f(e_1, f_1)
\end{align*}
\]

![Diagram showing P and Q with variables and their operands]

Example

SSA code
- \( x_0 = 1 \)
- \( y_0 = 2 \)
- \( x_1 = x_0 + 1 \)
- \( y_1 = y_0 + 1 \)
- \( z_1 = x_0 + 1 \)

Congruence classes
- \( S_0 = \{x_0\} \)
- \( S_1 = \{y_0\} \)
- \( S_2 = \{x_1, y_1, z_1\} \)
- \( S_3 = \{x_1, z_1\} \)
- \( S_4 = \{y_1\} \)

Worklist: \( S_0 = \{x_0\}, S_1 = \{y_0\}, S_2 = \{x_1, y_1, z_1\} \)
- \( S_0 \text{ psplit } S_0? \text{ no} \)
- \( S_0 \text{ psplit } S_1? \text{ no} \)
- \( S_0 \text{ psplit } S_2? \text{ yes!} \)

\( S_2 \setminus S_0 = \{x_1, z_1\} = S_3 \)
\( S_2 / S_0 = \{y_1\} = S_4 \)
Algorithm

Worklist $\leftarrow \emptyset$

For each function $f$

$C_f \leftarrow \emptyset$

For each assignment of the form “$x = f(a,b)$”

$C_f \leftarrow C_f \cup \{ x \}$

Worklist $\leftarrow$ Worklist $\cup \{ C_f \}$

CC $\leftarrow$ CC $\cup \{ C_f \}$

While Worklist $\neq \emptyset$

Delete some $D$ from Worklist

For each class $C$ properly split by $D$ (at operand $i$)

CC $\leftarrow$ CC $-$ C

Worklist $\leftarrow$ Worklist $-$ C

Create new congruence classes $C_j \leftarrow \{ C \setminus \downarrow D \}$ and $C_k \leftarrow \{ C / \downarrow D \}$

CC $\leftarrow$ CC $\cup$ $C_j$ $\cup$ $C_k$

Worklist $\leftarrow$ Worklist $\cup$ $C_j$ $\cup$ $C_k$

Note: see paper for optimization

Comparing Optimistic and Pessimistic

Differences

– Handling of loops
– Pessimistic makes worst-case assumptions on back edges
– Optimistic requires actual contradiction to split classes
Role of SSA

Single global result
– Single def reaches each use
– No data (flow value) at each point

No data flow analysis
– Optimistic: Iterate over congruence classes, not CFG nodes
– Pessimistic: Visit each assignment once

ϕ-functions
– Make data-flow merging explicit
– Treat like normal functions after subscripting them for each merge point

Concepts

SSA construction
– Place phi nodes
– Variable renaming

Transformation from SSA to executable code depends on the optimizations dead-code elimination and coalescing in the register allocator to remove extra moves

Some optimizations that are simpler and more efficient with SSA
– dead-code elimination
– constant propagation
– copy propagation
– value numbering

Others that weren’t covered
– induction variable detection, strength reduction, and elimination
– register allocation
– ...

Next Time

Lecture
- Parallelism and Data Locality, start reading chapter 11 in dragon book

Suggested Exercise
- Use any programming language to write the class declarations for the equivalence class and variable structures described at the bottom of page 6 in the value numbering chapter.
- Instantiate the data structures for the small example programs in Figures 7.2 and 7.6 and perform pessimistic value numbering. Remember to convert to SSA first.
- After performing value numbering, how will the program be transformed?